TRaDe, A Topological Approach to On-the-fly Race Detection in Java Programs

Mark Christiaens
ELIS, Ghent University
Gent, 9000, Belgium
mchristi@elis.rug.ac.be

Koen De Bosschere
ELIS, Ghent University
Gent, 9000, Belgium
kdb@elis.rug.ac.be

Abstract

Debugging multi-threaded programs is notoriously hard. Probably the worst type of bug occurring in multi-threaded programs is a data race. There is therefore a great need for tools to automatically detect data races during execution. This article presents, TRaDe, a novel approach to detect races in object-oriented languages using a topological approach. An implementation of TRaDe based on the Sun JVM 1.2.1 is compared with existing tools. TRaDe proves to be a factor 1.6 faster than any known race detection tool for Java and has memory requirements similar to the best competing tools.

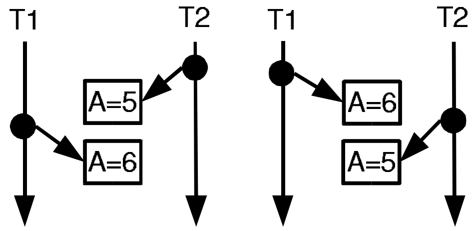

Figure 1: On the left we see thread 2 accessing a common object, `A`, and writing the value `5`. This is followed by thread 1, writing the value `6` to `A`. The result of this operation is that `A` contains the value `6`. On the right, thread 1 for some reason executes faster which results in the same events happening but in reverse order. This is possible since there is no synchronisation between thread 1 and 2. `A` now contains the value `5`.

1 Introduction

Multi-threaded applications are hard to debug. This is due to the fact that when searching bugs in multi-threaded applications we have to reason about a multitude of threads, each simultaneously performing separate tasks. One particularly hard bug to detect is a 'data race'. A data race occurs when the programmer does not correctly synchronize the access to a variable which is being manipulated by more than one thread. This can leave the variable in an unexpected or inconsistent state. In Figure 1, we see a simple example.

Data races are very hard to find because of two reasons. First of all, they are non-deterministic because they depend on the interleaving of the actions of threads, which is not always the same. Even if we observe them in one run, during a next run, they may not occur again, leaving us totally in the dark as to what went wrong. Secondly, they are non-local. One thread may be performing a spelling check and another may be editing the text being checked. These are two almost totally unrelated sections of code that, if not well synchronised, will cause havoc.

There are three major approaches to finding data races:

- Static analysis of parallel programs has been proven in general to be an NP hard problem [13].
- Post-mortem analysis usually involves large traces of the execution of the multi-threaded program [4, 14, 15] but recently new techniques were developed that make this approach viable [16, 17].
- On-the-fly analysis [5] has no need for traces since it tries to detect data races as they occur. However, it has the potential to be very intrusive which must be avoided as much as possible.

Current on-the-fly techniques incur large overheads due to the fact that they must observe every read and write operation to shared variables. Time overheads as high as a factor 30 are not uncommon.

In this article, we present, TRaDe (Topological RAce Detection), a novel method to automatically detect data races on-the-fly with reduced effort in "pure" object-oriented environments. Using this technique, we are able to dynamically make a selection of the objects we need to observe to find data races by analyzing the graph formed by the interconnection between these objects. This approach is applicable to a wide range of object-oriented languages. Since Java is widely used, object-oriented and multi-threaded, we will give a practical implementation as proof of concept by extending a Java Virtual Machine. We have compared TRaDe to two commercial competitors, JProbe [9] and AssureJ [10]. We have found that AssureJ ignores a subset of data races which we correctly detect. More importantly, TRaDe is on average a factor 1.6 faster than its closest competitors with comparable memory requirements.

In Section 2, we briefly describe the synchronisation primitives of Java and in this context we give our definition of data races. In Section 3, we present the idea of topological race detection followed by a description of our implementation in Section 4. Performance measurements are provided in Section 5. Finally, we indicate some avenues for future research in Section 6 and present our conclusions in Section 7.

2 Synchronisation in Java

Java [2] is an object-oriented language that was designed with multi-threading in mind [11, 12, 21]. In Java there are only two fundamental data types: 'primitive types' and 'reference types'. Primitive types consist of booleans, integers, floats, ... Reference types contain a reference to an object or contain 'null'. These objects are created dynamically on the heap. A garbage collector is responsible for destroying them when they are no longer referenced [8]. Objects themselves contain primitive types or references.

A race between two (or more) threads occurs when they modify a member variable of an object in an unpredictable order. Races on variables on the stack are impossible since the stack can only be manipulated by the thread it belongs to.

To avoid data races, a programmer can force fragments of code running on different threads to execute in a certain order by adding extra synchronisation operations. Java offers several constructs that enforce extra synchronisation[1]:

- `start` and `join` which operate on `Thread` objects,
- locked objects,
- `synchronized` (static) member functions and
- `wait` and `notify(All)`.

The fragments of code of a thread that are separated from each other by a synchronisation operation are abstracted into the notion of 'events'. Notice that the synchronisation operations are not considered events themselves. The i^{th} event of thread T_t will be denoted by $e_{t,i}$. Two events, e_i and e_j, are said to be 'ordered', $e_i \rightarrow e_j$, if there exists a set of synchronisations that force event e_i always to occur before event e_j. A data race occurs when there is no set of synchronisations that force the events modifying a shared variable to occur in a fixed order.

TRaDe models the ordering of events by using a construct called a 'vector clock' as defined in [6, 7, 20]. Vector clocks are tuples of integers with a dimension equal to the maximum degree of parallelism (number of threads) in the application. The first event, $e_{t,0}$ of every thread T_t is assigned the vector clock, $VC(e_{t,0})$, with components

$$VC(e_{t,0})_j = \begin{cases} 0, & j \neq t \\ 1, & j = t \end{cases} \tag{1}$$

The value of the vector clock of the next event in a thread is calculated using the vector clocks of its preceding events. If event $e_{t,i}$ on thread T_t is guaranteed to occur after events $E = \{e_0 \dots e_n\}$, its vector clock is updated as follows

$$VC(e_{t,i})_j = \begin{cases} max(\{VC(e)_j | e : E\} \cup \{0\}), & j \neq t \\ max(\{VC(e)_j | e : E\} \cup \{0\}) + 1, & j = t \end{cases} \tag{2}$$

[1] There are a few other operations on `Thread` objects, which influence the execution of other threads but which we do not take into consideration since they are either being removed from the Java APIs or cannot be used to synchronize two threads: `destroy`, `interrupt`, `resume` and `stop`.

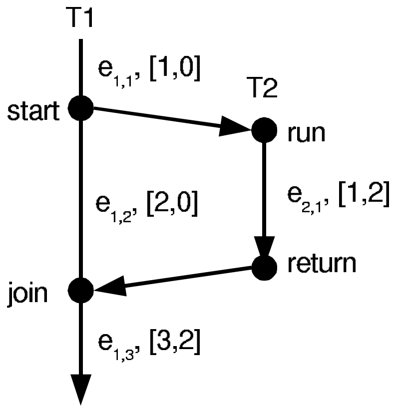

Figure 2: Synchronisation using the `Thread` class

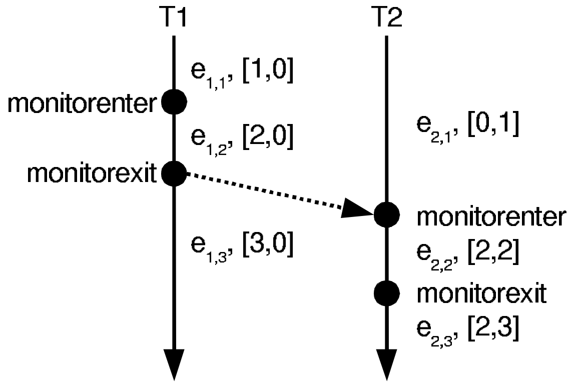

Figure 3: Synchronisation through a locked object

For our purposes, the most important property of vector clocks, is that they can be used to verify whether two events are ordered. Two events, a and b, are ordered iff

$$a \to b \equiv (\forall i. VC(a)_i \leq VC(b)_i) \wedge a \neq b \qquad (3)$$

Two events are parallel, i.e. not ordered, iff

$$a \parallel b \equiv \neg(a \to b) \wedge \neg(b \to a) \qquad (4)$$

If we define $W(a)$ the set of all locations written to during event a and $R(a)$ the set of all locations read during event a, then two events, a and b, will be involved in a data race iff

$$(a \parallel b) \wedge \begin{array}{l} (W(a) \cap R(b) \neq \emptyset) \vee \\ (R(a) \cap W(b) \neq \emptyset) \vee \\ (W(a) \cap W(b) \neq \emptyset) \end{array} \qquad (5)$$

i.e. the two events are executed in parallel, both events access a common variable and at least one event modifies the variable.

The ordering between events is obtained by observing the synchronisation operations in Java. The `start` member function of `Thread` is used by one thread to start the execution of a second thread. The `join` member function allows one thread to wait for the end of the execution of a second thread. These operations impose an ordering on the events of these threads as can be seen in Figure 2. The value of the vector clocks is shown in the figure, illustrating the calculation of the vector clocks at synchronisation operations.

A lock is associated with every `Object` in Java. A thread can try to take this lock using the bytecodes `monitorenter` and `monitorexit`. When the object is already locked, the thread will wait until it is unlocked and can then proceed. This construct does not impose a fixed time order on the code of the two threads involved, it just enforces mutual exclusion. It does suggest that the programmer is aware of a potential race and has used this construct as a means of synchronisation. We therefore consider this a 'de facto' ordering, depicted in Figure 3 by a dashed arrow.

The `synchronized` keyword is applied to a subset of the member functions of a class, the 'monitor'. When a thread invokes one of these member functions on an object of the synchronised class, Java ensures that none of the other member functions in the monitor is being executed. This is implemented through the object locking mechanism mentioned above. When a synchronised member function is executed, the lock of the object containing the member function is taken. When the member function finishes, the lock is released.

A final set of synchronisation primitives is `wait` and `notify(All)` which are member functions of every `Object`. When a thread invokes `wait` on an object, the execution of the thread is halted until another thread executes `notify(All)` on that very same object. At that time the first thread in line can continue its execution. This imposes the ordering depicted by the dotted arrow in Figure 4. However, a thread is only allowed to invoke `wait` or `notify` on an object if that thread owns the lock of that object, so in reality it suffices to observe the ordering between the `monitorenter` and `monitorexit` depicted by the solid arrows.

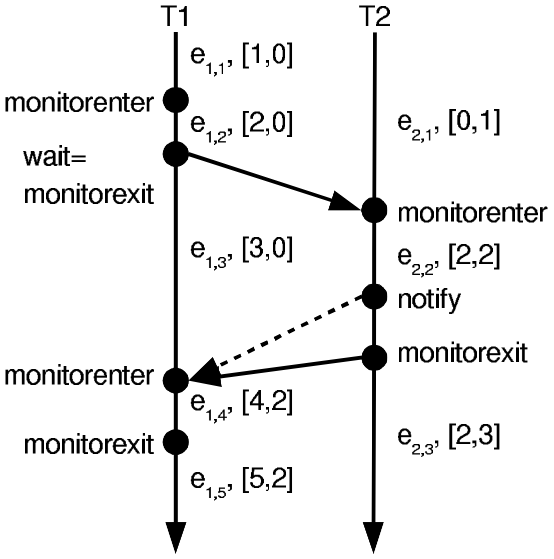

Figure 4: Synchronisation through signals

To detect data races, an access history for every object is constructed. In this access history, every read an write operation to the object must be stored together with the identity of the thread that performed the operation and the vector clock of the event to which the operation belongs. When a new read or write operation occurs, it is compared to the previous operations stored in the access history. If condition 5 is satisfied, a race is found.

Of course, as the program continues to execute, the access histories grow without bounds. In [5] it is shown that it suffices to store only the last[2] write operation in the access history, since these write operations must be ordered or a race would already have occurred. As a consequence, only one write operation is stored in the access history. Also, only the read operations which are parallel with each other need to be stored. So at most one read operation for every thread present in the program must be stored in the access history.

Still, this can amount to a very large overhead, especially if there are many threads (remember that the size of each vector clock grows proportionally to the number of threads). One way out is to reduce the number of objects for which an access history must be maintained. In the next section, we shall present a method which makes this possible.

[2]w.r.t. the vector clock ordering

3 Topological Race Detection

There are some apparent and also some less apparent advantages to doing race detection on the Java bytecodes instead of on the underlying hardware instructions.

First of all, we have the granularity argument. Since Java bytecodes are quite high level, many machine instructions can be necessary to perform one bytecode instruction. If we try to detect data races at the machine level, we will have to observe every hardware read/write instruction for every thread. If however we observe at the bytecode level, many machine instructions can safely be ignored, assuming a correct JVM.

Secondly, Java, by construction, makes a large number of instructions data race free. Every thread has its own private stack on which it allocates local variables and parameters for each member function call. This data can only be modified by the owning thread. As a consequence, all instructions that solely manipulate stack data cannot cause a data race. There are 181 guaranteed data race free instructions and 20 'dangerous' instructions. The latter set can be split into 2 categories. The first category consist of the {`a,b,c,d,f,i,l,s`}`aload` and {`a,b,c,d,f,i,l,s`}`astore` bytecodes. These instructions read and write the contents of arrays on the heap. Since objects on the heap can be reached from multiple threads, these instructions need to be checked for data races. The second category consists of `getfield`, `getstatic`, `putfield` and `putstatic`. These are instructions that read or write the fields of objects or classes on the heap.

Finally, and this will prove essential to our technique, Java enforces a very strict object model, even on the bytecode level. Machine instructions can modify practically any location in the address space of a program. Java, on the other hand, only allows modification of an object's data through the reference to that object.

As can be seen in Figure 5, an object is created by invoking the bytecodes `new`, `newarray`, `anewarray` or `multianewarray` (1 in the figure). These instructions all put a reference to the newly allocated object on the stack of the invoking thread. At this point, these objects are only reachable through this one reference and this reference is only accessible to one thread. We call this situation a 'local object'.

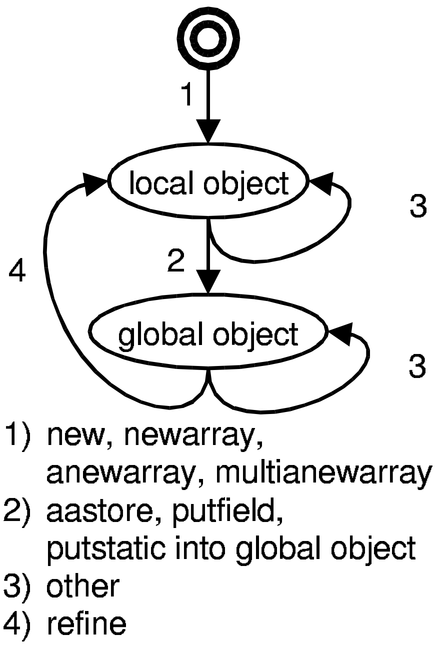

Figure 5: The life cycle of an object

No races are possible. An object that is reachable, through some path, by several threads, is called a 'global object'. It has the potential to be involved in a race. Since we construct information about the reachability of objects from several threads in order to detect potential race problems, we call this method 'topological race detection', TRaDe for short.

Initially an object is created locally. The only references to it exist on its creating thread's stack (1 in Figure 5). One way to change the status of an object from local to global is by storing its reference into a second object. If this second object is reachable by another thread, our object also becomes reachable by this other thread (2 in Figure 5). At this point, the object could potentially be involved in a race. When an object becomes global all the objects reachable from this new global object also become global. If, on the other hand, the second object is solely reachable by our own thread, it remains local.

There are only a small number of bytecodes that can change the topology of the object interconnection graph: `aastore`, `putfield` and `putstatic`. They all have in common that they store references in an object's field.

There are a few exceptions to the outline given above. Every object of type `Class` is global right from the start. The reason for this is that every thread needs to be able to access every class to construct objects of this class. Inside a class, there are static variables that can be read and written to. So these are, by definition of the Java language, immediately global to all threads.

A second way in which an object can become global is when it is involved in the startup of a new thread. In Java, threads are started by creating an object containing a `run` method. When this object's `start` method is called, a new thread is created and starts executing the code in the `run` method. At this very point, this object is reachable by both the new thread and the original thread that started the new thread. It must therefore immediately be made global together with all objects reachable from this object.

This last observation is crucial and is the reason why a refinement of our global objects is necessary (5 in Figure 5). Consider for example the Java program in Figure 6. It creates 10 separate threads (besides the `main` thread). Each of these threads creates a linked list of 10000 local objects.[3] No races are present in the program. Note that the main thread does not maintain a link to the started threads (`g = null`).

This program was artificially made to be very suitable for our approach to race detection; it is very simple and contains large data structures which are clearly not shared between the threads. Almost all calculations are performed on these local data structures so fairly little overhead should be incurred by doing race detection. Still, without the refinement step, TRaDe would perform very badly.

When the new threads are started (at line 29) by invoking `start` on the object `g` of the class `Separate`, `g` must be made global. If it were not for the refinement, these objects would remain global for the remainder of the program's execution. When the threads would start to construct their linked lists, these linked list would also become global. Not much can be gained from such an approach. To do the refinement step, we turned to the garbage collector (GC).

Whatever the underlying algorithm of the GC is, somehow it must determine whether an object is no

[3] Of course the JVM makes hundreds more objects for its internal use but this is not important in this example.

```
class Separate extends Thread {
  Link root;
  int count;

  class Link { Link prev, next; }

  Separate (int count) {
    root = new Link ();
    root.prev = root.next = root;
    this.count = count;
  }

  public void run () {
    for (int i = 0; i < count; i++) {
      Link l = new Link ();

      l.prev = root;
      l.next = root.next;
      root.next.prev = l;
      root.next = l;
    }
  }

  public static void main (String [] arg) {
    Separate g;

    for (int i = 0; i < 40; i++) {
      g = new Separate (10000);
      g.start ();
    }

    g = null;
  }
}
```

Figure 6: The test program

longer reachable by any thread in the program. If this is the case, the object can be removed from the heap. This is very similar to what TRaDe is trying to do. TRaDe tries to determine whether an object is reachable by *more* than one thread in the program.

We exploited this observation as follows. Each time the GC performs its job, it is followed by our 'refiner'. For every thread, we generate a set, S_i, of all the objects that are reachable from that thread. Then we combine these sets into a set of objects reachable from multiple threads as follows.

$$S_{tot} = \cup_{i,j}(S_i \cap S_j) \tag{6}$$

We use S_{tot} to refine the general TRaDe mechanism after garbage collection. If an object is not present in S_{tot}, it is only reachable from one thread and therefore local. The large data structures that are necessary to enable data race detection are removed and the object is marked as being reachable only by this one thread.

A minor, yet crucial, modification to our refiner proved necessary. A reference to each `Thread` (and derived classes) is always present in at least one `ThreadGroup` and every `Thread` can obtain the `ThreadGroup` of which it is a member. This has the annoying consequence that every `Thread` can be reached from every other `Thread` through its `ThreadGroup`. If we instruct our refiner to only collect the objects reachable from one thread, it will inevitably start collecting a large portion of the other thread's data structures through these hidden references. The solution was not to follow links leaving `ThreadGroup`. Should a program use these references in `ThreadGroup`, it will circumvent our technique. We believe this will occur only rarely. If this is a problem, we could adapt the JVM further so as to flag such behavior.

Figure 7 compares the approach without a refinement phase (top graph) with the approach with a refinement phase (bottom graph). The figures are constructed by analyzing the heap each time a garbage collection has occurred. We see the total number of handles allocated (a handle points to an object). These handles are subdivided into unused handles which are preallocated each time the heap is expanded, local handles which point to objects that are detected as being local to a thread and global handles which point to objects which were detected as being reachable by more than one thread.

The top graph shows that practically all used handles point to global objects. This means that we would have to observe accesses to most objects for potential data races. The number of local objects is so small that they are not noticeable on the graph.

The bottom graph, using the refiner, shows the expected result. We see that almost all used handles point to local objects. The number of global objects is very small. Clearly, the large linked lists are being detected as local so we will not need to observe these to detect data races.

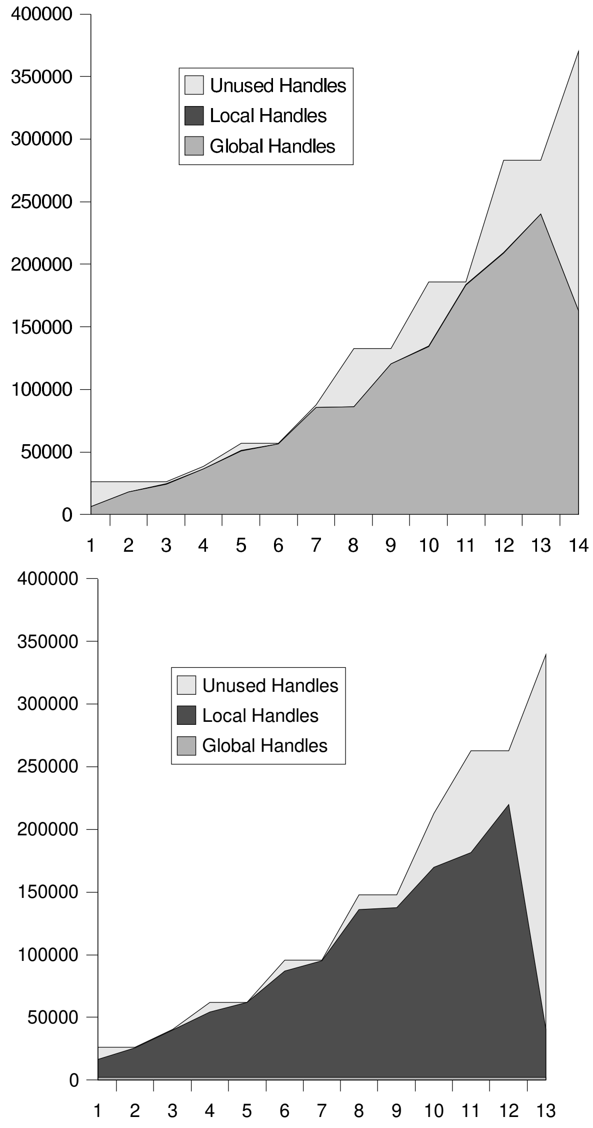

Figure 7: Total number of handles in use measured after each garbage collection phase. The top graph was measured without using the refiner. The bottom graph was measured while using the refiner.

4 Implementation

To test our ideas in practice, we implemented the TRaDe method in an existing JVM. We selected the Sun JVM 1.2.1 on Solaris. This JVM comes equipped with a just in time compiler (JIT). A JIT is used to compile bytecodes to machine instructions on-the-fly so as to accelerate the execution of a Java program. We decided to turn the JIT compiler off to simplify our coding. This way, the JVM just uses an interpreter loop, executing the bytecodes one by one. Our techniques should be readily transferable to JIT compilers.

Our first step was to instrument all the synchronisation primitives of Java using vector clocks as described in Section 2. Vector clocks have a serious drawback: they contain as many components as there are threads in the program. This means that if we are dealing for example with an FTP-server which creates a thread for every file request it receives, the size of the vector clock grows without bounds. We have implemented an advanced version of vector clocks that can dynamically grow *and* shrink as threads are created and destroyed without losing accuracy while doing data race detection and with little overhead involved called ‘accordion clocks’. The exact approach we took is beyond the scope of this article.

The next step was to instrument every object with a minimal data structure that allows TRaDe to be used. The basic idea can be seen in Figure 8. When objects are created using `new`, `newarray`, `anewarray` or `multianewarray`, they are extended with a data structure consisting of 20 extra bytes.[4] It consists of 2 parts. The first is the thread identification number (`TID`). In this field, the `TID` of the thread that created this object is stored or, when the object becomes global and is reachable by several threads, -1 is stored. The second part consists of link fields that will be used to link a much larger data structure for full data race detection *only* if the object becomes global.

An object can contain several fields that can be written or read (`#fields`). If we instrument a new global object, each field must have its specific data structure that maintains its access history. This data structure contains: a description (`description`) of the field being accessed (its name,

[4]This could be reduced to 8 bytes, but this is just a prototype.

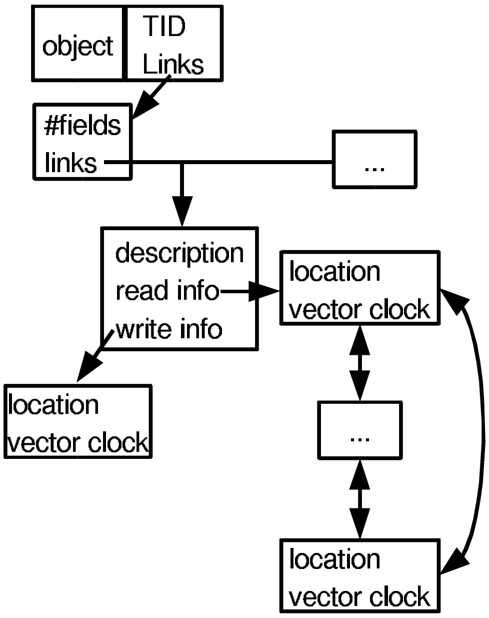

Figure 8: Full instrumentation of an object for data race detection

type info, ...) and links to information about the read and write operations that involved this field. For each read and write operation we store the location in the code where the operation occurred (a class, a member function and a JVM program counter) and a vector clock indicating 'when' the operation occurred.

Using this data structure, the instructions `aastore`, `putfield` and `putstatic` are instrumented. We will explain what happens for `aastore`. A similar procedure is followed for `putfield` and `putstatic`.

Suppose the bytecode `aastore` stores a reference, `R`, into an array, referred to by reference `A`. There are 2 possibilities:

- If the object pointed to by `R` is already global (`R.TID == -1`) then nothing happens, the object is already being watched for possible data races.
- If on the other hand, the object is not yet global, the `TID` of the array referred to by `A` is checked. If it is global (`A.TID == -1`), then by storing `R` into `A`, the object referred to by `R` also becomes global. Otherwise, if `A.TID != R.TID`, we are storing our reference into an array that is reachable by another thread. The object referred to by `R` must again be made global.

If the object referred to by `R` becomes global, we recursively check all its children. Each child that is not yet global is made global. Here we must pay attention to stack overflow when recursively marking a deep data structure as global.

The actual race detection consists of instrumentation added to the 20 bytecodes which read or write to an object as follows. Each time such a bytecode is executed, we check whether it affects a global object. If not, we don't have to do anything; races are impossible. If we are dealing with a global object, we can access the extra data structures and verify using the vector clocks whether this new instruction represents a data race. If so, we flag this to the user. Then we update the access history with the new location of this instruction and the new vector clock indicating when the instruction occurred.

5 Performance Measurements and Comparison with Existing Tools

A number of general tools have already been developed to automatically detect data races in a program. Eraser[5], for example verifies the locking discipline [19]. If a memory location is read/written by different threads, a set of locks must be held. Each time this location is accessed again, the tool checks which locks are held and whether their use is consistent with previous use. Another tool for checking, among others, for data races is RecPlay [17]. It takes a different approach from Eraser's since it performs data race detection off-line, using a recorded trace from a previous execution. The types of races detected are similar to our definition given above.

Both these tools function 'blindly', not knowing what type of program they are analyzing, and observe the stream of processor instructions that are being executed and the memory locations upon which they operate. In contrast, true Java specific tools also exist.

JProbe is a tool capable to detect, among other things, data races [9] in Java programs. It seems

[5]Eraser is apparently now marketed by Compaq under the name VisualThreads

to be using Sun's Java Virtual Machine Profiler Interface (JVMPI). This is an interface that allows profilers to request to be notified of certain events in a Java program such as the loading of classes, start of garbage collection, entering of monitors, etc. Although nothing is published about its internal workings, except its user manual, it seems to use a similar definition of data races as the one used by TRaDe.

Another Java tool is AssureJ [10]. It is also capable of data race detection, among other things, and is very fast. Nothing is known about the algorithm it uses. It does not seem to use the JVMPI but rather to be a modified Sun 1.2 JVM. Again, it seems to use a similar definition of a race as used in TRaDe. One important short-coming that was noticed is that when two events race (so their vector clocks are parallel) but their threads do not actually overlap in time, no race is detected.[6] This is probably due to the fact that they remove all information concerning the behaviour of a thread as soon as it terminates, which is incorrect.

We've run extensive benchmarks comparing TRaDe to JProbe and AssureJ. See Table 1 for a description of the benchmark programs and options used. We selected large applications from a wide variety of problem domains.

The options used for each race detection tool are configured so that each has to find races in the whole of the program. Both JProbe and AssureJ have options to focus their race detection on certain sections of the code. These options were turned off. All programs have the ability to detect races on arrays as a whole, called 'collapsing' arrays, or to consider the elements of an array as separate entities which each can be involved in a race. We chose to collapse arrays to conserve memory since this is the recommended setting for JProbe and AssureJ. Each was configured so as to enable their best performance. All other features were turned off.

The results can be seen in Table 2. All benchmarks were run on a Sun Ultra 5 workstation with 512 MB of memory and a 333 MHz UltraSPARC IIi with a 16 KB L1 cache and a 2MB L2 cache. Memory usage and user time were estimated by averaging 5 measurements with no other user programs running. Some benchmarks could not be run in 512 MB of memory. A larger system was then used as an indication of how much more resources are needed to complete the benchmark. In this case, a Sun Ultra-2 was used with 2048 MB of memory, 4X400 Mhz UltraSPARC IIi processors with 16 KB L1 direct mapped cache and 4MB L2 direct mapped cache each. Note that we were unable to use this machine exclusively. Other user programs were running simultaneously.

We also added some baseline measurements for comparison with normal execution without race detection. Figures using the Hotspot 1.0.1, mixed mode, build f were added. This is a state of the art just-in-time (JIT) compiler from Sun which compiles Java code to native code where necessary. Since we are not using the JIT included in the Sun JVM, we also added baseline figures using only the interpreter version of the JVM.

As can be seen, TRaDe is faster on all benchmarks than JProbe and AssureJ. It beats JProbe by a very large margin; many benchmarks cannot be completed due to memory exhaustion. When averaged out, TRaDe is a factor 1.6 faster than AssureJ. As to memory consumption, TRaDe again beats JProbe by a very large margin.

AssureJ is more on par with TRaDe in the area of memory consumption. AssureJ, on average, uses only a factor 0.74 of the memory TRaDe uses. This may be caused by the fact that AssureJ incorrectly removes information about dead threads but until they divulge their algorithm, it is anybody's guess.

6 Future Work

As can be seen from Table 2, the overhead of data race detection is still large when comparing to normal execution. We plan to further investigate whether it is possible to tighten the gap between normal execution of Java programs and our race detection through the use of a JIT compiler.

In TRaDe, we are using an advanced form of vector clocks, called 'accordion clocks', which dynamically grow and shrink. We will evaluate their performance in highly multi-threaded applications.

Recently, a static analysis technique, called 'escape analysis', has been applied to Java (see for example [1, 3, 18]). It is used to classify objects as escaping from a method or from a thread. Objects that

[6] This shortcoming was confirmed by their product support.

SwingSet Demo	A highly multi-threaded demo, included with the JDK 1.2, of the Swing widget set. It demonstrates buttons, sliders, text areas, ... The demo was run until it was fully loaded and displaying its initial screen. Immediately thereafter it was shut down.
Jess 6.01a1	Jess is a clone of the expert system shell CLIPS, rewritten entirely in Java. Input is provided so that is solves the famous 'The monkey and the banana' problem included with the Jess distribution. Run with: `jess.Main examples/fullmab.clp`
Resin 1.2b1	Resin is a web server entirely written in Java. It supports JSP, XML, JavaScript, XTP, XSL, ... The JSP files `hello.jsp`, `env.jsp`, `counter.jsp` and `index.jsp`, included in their examples directory, were requested in parallel.
Colt 1.0.1.56	Open Source Libraries for High Performance Scientific and Technical Computing in Java. A benchmark program is included in the distribution which does a number of matrix calculations. Run with: `cern.colt.matrix.bench.BenchmarkMatrix dgemm dense 10 2 0.99 false true 5 5 50 100 300 50 100`
Raja 0.4.0 pre2	Ray tracer in Java generating a picture demonstrating their Phong highlight capabilities. `raja.ui.Compute -v -p txt -r 128x128 -d20 -o Phong-128x128.png Phong.raj`

Table 1: Description of benchmark programs

	TRaDe		AssureJ		JProbe		Hotspot		Interpreter	
	s	MB	s	MB	s	MB	s	MB	s	MB
SwingSet	98.3	126.6	160.6	73.3	>1200 †	>650 †	20	41.8	15	29.8
Jess	370.3	12.1	610	17.5	>3600 †	>650 †	22	19.8	76	8
Resin	56.5	27.6	68	27.3	193	226.17	11.8	27.9	10	13.7
Colt	132.5	25	187.8	21.2	471.6	71	27.8	23.6	40.4	13.5
Raja	204.8	19.5	372	17.8	1945 †	1037 †	14.6	24.5	42.2	11.3

Table 2: Performance measurements

remain local to a method can be stack allocated, removing the overhead of heap allocation. Synchronisation operations can be removed when operating on objects that are local to a thread. Although this is a static, conservative technique, not applicable to Java programs dynamically loading classes, it would be interesting to have a clearer view on how many objects escape analysis can mark as thread local in comparison to TRaDe.

7 Conclusions

Data races are a serious problem inherent to multithreaded programs. There exist a number of general techniques to combat data races but they are slow and very intrusive. In this article, we have introduced a new technique, TRaDe, that uses information about the changing topology of the objects interconnection graph to more efficiently detect data races. Despite the fact that TRaDe does not cut corners while doing data race detection, it is faster than all known competition and comparable to the best in memory usage.

8 Acknowledgements

Mark Christiaens is supported by the IWT SEESCOA project (IWT/ADV/980.374) Our thanks go to Michiel Ronsse, with whom we had many stimulating discussions and to professor Jan Van Campenhout for many valuable corrections to this article.

References

[1] J. Aldrich, C. Chambers, E. G. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *Static Analysis Symposium 99*, September 1999.

[2] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, Massachusetts, USA, 1996.

[3] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronisation in Java . Technical report, Department of Computer Science, University of California, Santa Barbara, CA 93106, april 1999.

[4] Jong-Deok Choi, Barton P. Miller, and H. B. Netzer, Robert. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, October 1991.

[5] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 1–10, March 1990.

[6] C.J. Fidge. Partial orders for parallel debugging. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and distributed debugging*, pages 183–194, May 1988.

[7] Dieter Haban and Wolfgang Weigel. Global events and global breakpoints in distributed systems. In *21st Annual Hawaii International Conference on System Sciences*, volume II, pages 166–175. IEEE Computer Society, January 1988.

[8] Richard Jones and Rafael Lins. *Garbage Collection, Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons, 1996.

[9] KL Group, 260 King Street East, Toronto, Ontario, Canada. *Getting Started with JProbe.*

[10] Kuck & Associates, Inc., 1906 Fox Drive, Champaign, IL 61820-7345, USA. *Assure User's Manual*, 2.0 edition, march 1999.

[11] Doug Lea. *Concurrent Programming in Java.* The Java Series. Addison-Wesley, Reading, Massachusetts, USA, 1997.

[12] Time Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, Reading, Massachusetts, USA, second edition edition, April 1999.

[13] R. H. B. Netzer. *Race Condition Detection for Debugging Shared-Memory Parallel Programs.* PhD thesis, University of Wisconsin-Madison, 1991.

[14] Robert H. B. Netzer and Barton P. Miller. Improving the accuracy of data race detection. In *Proceedings of the third ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 133–144.

[15] Robert H. B. Netzer and Barton P. Miller. Experience with techniques for refining data race detection. Technical Report CS-92-55, Dept. of Computer Science, Brown University, Department of Computer Science, Brown University, Providence, Rhode Island 02912, November 1992.

[16] Michiel Ronsse. *Racedetectie in Parallelle Programma's door Gecontroleerde Heruitvoering*. PhD thesis, Universiteit Gent, May 1999.

[17] Michiel Ronsse and Koen De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, May 1999.

[18] Erik Ruf. Effective synchronization removal for java. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 208–218. ACM, June 2000.

[19] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Operating Systems Review*, volume 31, pages 27–37. ACM, October 1997.

[20] Reinhard Schwarz and Friedman Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[21] Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill, New York, New York, USA, second edition, 1999.